

Traveey Assignment

Below slides elaborates assignment task of building the backend API for a web application that will allow the company to manage their employees and tasks. The application store employees and tasks data in separate tables in a database and establish a one-to-many relationship between employees and their tasks by having common key attribute within them.

Structure of application :

```

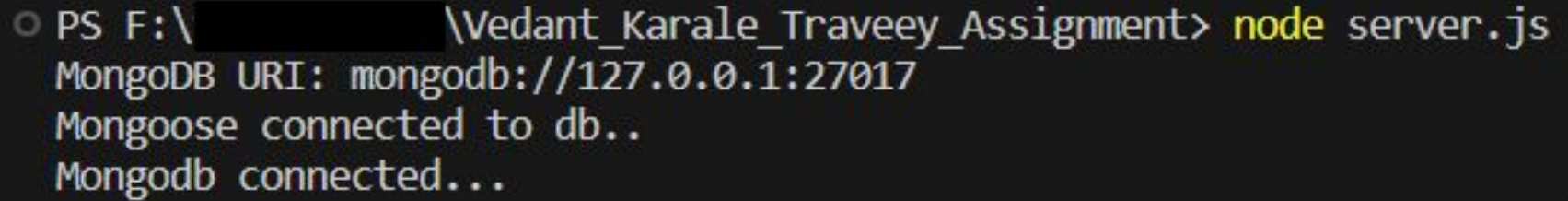
  ✓ Vedant_Karale_Traveey_Assignment
    ✓ Controller
      JS companyController.js
    ✓ Models
      JS employeeModel.js
      JS taskModel.js
    > node_modules
    ✓ Routes
      JS route.js
    ⚙ .env
    ⚙ .gitignore
    JS initDB.js
    {} package-lock.json
    {} package.json
    JS server.js
```

Entry Point for ExpressJs Application

```
1  const express=require('express')
2  const mongoose=require('mongoose')
3  const Routes=require('./Routes/route')
4  require('dotenv').config()
5
6  app=express()
7
8  app.use(express.json())
9
10 require('./initDB')()
11
12 app.get('/',(req,res)=>{
13   |   res.send("Welcome")
14   | })
15
16 app.use('/',Routes);
17
18 app.use((req,res)=>{
19   |   res.status(404).send("Error: Not Found")
20   | })
21
22 app.listen(3000)
```

This code sets up an Express.js application with basic routing and error handling. It connects to a MongoDB database using Mongoose and defines routes using an external module (Routes). When the server is started, it listens on port 3000 and responds to requests based on the defined routes and middleware. The actual route implementations are likely located in the Routes module, which is imported and mounted in this main application file.

As soon as the server is started by running 'node server.js', and if the application runs successfully, to ensure successful run of application, it will show the messages in the terminal about MongoURI and status of connection between Application and MongoDB database, as shown in image below, and will start listening on port 3000,

A terminal window with a black background and white text. The prompt is 'PS F:\Vedant_Karale_Traveey_Assignment>'. The command 'node server.js' has been executed. The output consists of three lines: 'MongoDB URI: mongodb://127.0.0.1:27017', 'Mongoose connected to db..', and 'Mongodb connected...'.

```
PS F:\Vedant_Karale_Traveey_Assignment> node server.js
MongoDB URI: mongodb://127.0.0.1:27017
Mongoose connected to db..
Mongodb connected...
```

Establishing connection between Application and MongoDB Database

```
1  const mongoose=require('mongoose')
2  const dotenv=require('dotenv').config()
3
4  console.log('MongoDB URI:', process.env.MONGO_URL);
5
6  module.exports=()=>{
7
8      mongoose.set('strictQuery', false);
9
10     mongoose.connect(process.env.MONGO_URL,{
11         dbName:process.env.DB_NAME,
12         user:'',
13         pass:'',
14         useNewUrlParser:true,
15         useUnifiedTopology:true
16     }).then(()=>{
17         console.log('Mongodb connected...')
18     }).catch(err=>console.log(err.message));
19
20     mongoose.connection.on('connected',()=>{
21         console.log("Mongoose connected to db..")
22     })
23     mongoose.connection.on('error',(err)=>{
24         console.log(err.message)
25     })
26     mongoose.connection.on('disconnected',()=>{
27         console.log("Mongoose connection is disconnected")
28     })
29
30     process.on('SIGINT',()=>{
31         mongoose.connection.close(()=>{
32             console.log("Mongoose connetion is disconnected due to app termination")
33             process.exit(0)
34         })
35     })
36 }
```

This code sets up a connection to a MongoDB database, provides event handlers to log connection-related events, and ensures a graceful exit when the application is terminated. It's called from entry point Node.js application module to establish a database connection at the start of the application.

Essentially, this code is responsible for setting up a connection to a MongoDB database using the Mongoose library and handling various database-related events.

Employee Model

```
1  const mongoose=require('mongoose')
2  const schema=mongoose.Schema
3
4  const employeeSchema=new schema({
5
6      ID:{
7          type:Number,
8          required:true,
9          unique:true
10     },
11     name:{
12         type:String,
13         required:true
14     },
15     email:{
16         type:String,
17         required:true,
18         unique:true
19     },
20     phone:{
21         type:String,
22         required:true
23     },
24     hiredate:{
25         type>Date,
26         required:true
27     },
28     position:{
29         type:String,
30         required:true
31     }
32 })
33
34 const Empolyee=mongoose.model('employee',employeeSchema)
35 module.exports=Empolyee
```

This code defines a Mongoose schema and model for an "employee" in a MongoDB database.

In summary, this code defines a Mongoose schema and model for storing and managing employee data in a MongoDB database. The schema defines the structure of an employee document, specifying the data types and constraints for each field. The model allows you to create, read, update, and delete employee records in the database using Mongoose's API.

Task Model

```
1  const mongoose=require('mongoose')
2  const schema=mongoose.Schema
3
4  const taskSchema=new schema({
5      ID:{
6          type:Number,
7          required:true,
8          unique:true
9      },
10     title:{
11         type:String,
12         required:true
13     },
14     description:{
15         type:String,
16         required:true
17     },
18     duedate:{
19         type>Date,
20         required:true
21     },
22     employeeId:{
23         type:Number,
24         required:true
25     }
26 })
27
28 const Task=mongoose.model('task',taskSchema)
29 module.exports=Task
```

This code defines a Mongoose schema and model for a "task" in a MongoDB database.

In summary, this code defines a Mongoose schema and model for storing and managing task data in a MongoDB database. The schema defines the structure of a task document, specifying the data types and constraints for each field. The model allows you to create, read, update, and delete task records in the database using Mongoose's API. Each task document will be stored in a MongoDB collection named "task."

Client-Application Routes handled by Controller

```
1  const express=require('express')
2  const route=express.Router();
3  const controller=require('../Controller/companyController')
4
5  const Task=require('../Models/taskModel')
6  const Employee=require('../Models/employeeModel')
7
8  route.get('/employees',controller.GetAllEmployee)
9
10 route.post('/employees',controller.AddEmployee)
11
12 route.get('/employees/:id',controller.GetsingleEmployee)
13
14 route.patch('/employees/:id',controller.UpdateEmployee)
15
16 route.delete('/employees/:id',controller.DeleteEmployee)
17
18 route.get('/employees/:id/tasks',controller.GetAllTask)
19
20 route.post(['/employees/:id/tasks',controller.Addnewtask])
21
22 module.exports=route
```


Node.js code in above screenshot defines a set of routes using Express.js Router (`express.Router()`) for handling employee-related data and tasks. These routes correspond to various CRUD (Create, Read, Update, Delete) operations for employees and tasks.

As soon as the client sends request at server side, server invokes appropriate function present in controller module based on specified path i.e. controller will then handles requests and performs appropriate function, further the communication with database is also handled by controller

These routes serve as an interface to interact with employee and task data in the application. The actual logic for handling these routes is implemented in the 'companyController' module, which is responsible for processing the incoming requests, interacting with the database models, and sending responses to the client.

Controllers -

- **Controller for fetching all employees**

```
GetAllEmployee:async(req,res)=>{
  try{
    const results=await Employee.find({}, {_v:0});
    if(!results)return res.send('No Employee found')
    res.send(results)
  }catch(err){
    res.status(500).send('Error')
  }
},
```

This asynchronous function handles an HTTP GET request to retrieve all employee records.

It uses `await Employee.find({}, {_v: 0})` to query the MongoDB database for all employee documents, excluding the "`__v`" field.

If no employees are found, it sends a "No Employee found" response.

If employees are found, it sends the employee data as a response.

- **Controller for adding employee to database**

```
AddEmployee:async(req,res)=>{  
  try{  
    const employee=new Employee(req.body);  
    const result=await employee.save()  
    res.send(result)  
    console.log("Employee added")  
  }catch(err){  
    res.status(500).send('unable to add employee')  
  }  
},
```

This asynchronous function handles an HTTP POST request to add a new employee record.

It creates a new Employee instance using new Employee(req.body) with the request body data.

It then saves the new employee to the database using await employee.save().

If the employee is successfully added, it sends the newly created employee data as a response and logs "Employee added."

- **Controller for fetching single employee to database based employee id added**

```
GetSingleEmployee:async(req,res)=>{
  const sid=req.params.id
  try{
    const employee=await Employee.find({ID:sid})
    if(!employee)return res.status(500).send('Not found');
    res.send(employee)
  }catch(err){
    res.status(500).send(err)
  }
},
```

This asynchronous function handles an HTTP GET request to retrieve a single employee record based on the provided id parameter.

It extracts the id parameter from the request using `const sid = req.params.id`.

It uses `await Employee.find({ID: sid})` to query the database for an employee with the specified ID.

If no employee is found, it sends a "Not found" response.

If an employee is found, it sends the employee data as a response.

- **Controller for updating already existing employee information**

```
UpdateEmployee:async(req,res)=>{
  try{
    const updateId=req.params.id;
    const update=req.body;
    if(!update)return res.status(404).send('invalid update');
    const options={new:true}
    const update_entry=await Employee.findOneAndUpdate({ID:updateId},update,options)
    if(!update_entry){
      return res.status(500).json({message:'NO such task'})
    }
    res.send(update_entry)
  }catch(err){
    res.status(500).send(err)
  }
},
```

This asynchronous function handles an HTTP PATCH request to update an employee record based on the provided id parameter.

It extracts the id parameter from the request using `const updateId = req.params.id`.

It expects an updated employee object in the request body.

It checks if the update object exists; if not, it sends a "invalid update" response.

It uses `await Employee.findOneAndUpdate({ID: updateId}, update, options)` to find and update the employee record.

If no employee is found for the provided ID, it sends a "No such task" response.

If the employee is successfully updated, it sends the updated employee data as a response.

- **Controller for fetching all tasks based on employee id**

```
GetAllTask:async(req,res)=>{
  const employeeID=req.params.id;
  try{
    const results=await Task.find({employeeId:employeeID},{_v:0});
    if(!results)return res.send('No task found')
    res.send(results)
  }catch(err){
    res.status(500).send(err)
  }
},
```

This asynchronous function handles an HTTP GET request to retrieve all tasks associated with a specific employee based on the provided id parameter.

It extracts the id parameter from the request using `const employeeID = req.params.id`.

It uses `await Task.find({employeeId: employeeID}, {_v: 0})` to query the database for tasks associated with the specified employee ID.

If no tasks are found, it sends a "No task found" response.

If tasks are found, it sends the task data as a response.

- **Controller for adding new task to database**

```
AddNewTask:async(req,res)=>{
  try{
    const task=new Task(req.body);
    const result=await task.save()
    res.send(result)
    console.log("Task added")
  }catch(err){
    res.status(500).send(err)
  }
},
```

This asynchronous function handles an HTTP POST request to add a new task associated with a specific employee based on the provided id parameter.

It expects task data in the request body.

It creates a new Task instance using new Task(req.body) with the request body data.

It then saves the new task to the database using await task.save().

If the task is successfully added, it sends the newly created task data as a response and logs "Task added."

- **Controller for deleting employee from database**

```
DeleteEmployee:async(req,res)=>{  
  try{  
    const deleteid=req.params.id;  
    const result=await Employee.findOneAndDelete({ID:deleteid})  
    if(!result)return res.status(500).send('Task not found');  
    res.send(result)  
  }catch(err){  
    res.status(500).send(err)  
  }  
}
```

This asynchronous function handles an HTTP DELETE request to delete an employee record based on the provided id parameter.

It extracts the id parameter from the request using `const deleteld = req.params.id`. It uses `await Employee.findOneAndDelete({ID: deleteld})` to find and delete the employee record.

If no employee is found for the provided ID, it sends a "Task not found" response.

If the employee is successfully deleted, it sends the deleted employee data as a response.

After application run and connected with MongoDB database successfully, requests for different operations can be sent

For adding (POST) employee to database POST method is selected, server URI is provided and in body JSON object is provided containing all required information fields and with all fields in appropriate datatypes, using any API client software THUNDER CLIENT in this case, setting all this appropriately will successfully add employee to database

The screenshot displays the Thunder Client interface with a POST request to `http://localhost:3000/employees`. The request body is a JSON object representing an employee. A callout box points to the phone number field, indicating it should be entered. The response shows a successful status (200 OK) and a JSON object with the created employee's details.

Request:

```
POST http://localhost:3000/employees
```

JSON Content:

```
{
  "ID": 1,
  "name": "John Doe",
  "email": "john@email.com",
  "phone": "98765",
  "hiredate": "2015-03-25",
  "position": "Software Developer"
}
```

Response:

```
{
  "ID": 1,
  "name": "John Doe",
  "email": "john@email.com",
  "phone": "98765",
  "hiredate": "2015-03-25T00:00:00.000Z",
  "position": "Software Developer",
  "_id": "65102ed7d1c005de6063e435",
  "__v": 0
}
```

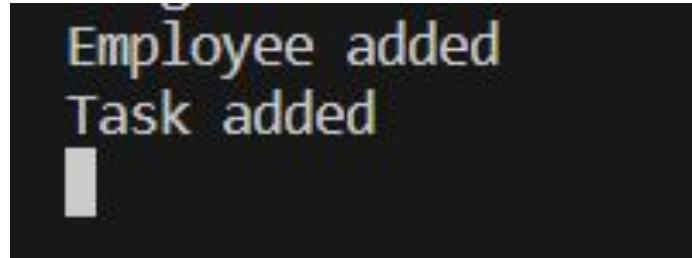
Employee entity will be added into database in employees collection and task entity will be added into database in tasks collection of employee database.

There will be one-to-many relationship between employee entity and task entity such that each employee can be having many tasks associated with it and each task will be associated with one employee only, this is implemented by adding employeeId field in task entity as a foreign key.

Other operations can be done by following below syntaxes, where id will be replaced with required employee and task id respectively :

- GET /employees: Retrieves a list of all employees
- POST /employees: Creates a new employee
- GET /employees/:id: Retrieves a specific employee by their id
- PATCH /employees/:id: Updates a specific employee by their id
- DELETE /employees/:id: Deletes a specific employee by their id
- GET /employees/:id/tasks: Retrieves all tasks associated with a specific employee by their id
- POST /employees/:id/tasks: Creates a new task for a specific employee by their id

After each successful addition of employee entity and task entity, below message is displayed in terminal and ensures if each employee and task is added successfully

A screenshot of a terminal window with a black background. It displays two lines of text in a yellow, monospaced font: 'Employee added' on the first line and 'Task added' on the second line. A white cursor is positioned at the start of the third line.

```
Employee added  
Task added  
█
```

Setup and run application locally

- Get project from Git Repository to local system
- Open project with any compatible IDE (preferably VS Code editor, some of the below steps can be slightly relevant to VS Code editor)
- Below usages are present in .env file of application
- DB_NAME=employee
- MONGO_URL=mongodb://127.0.0.1:27017
- Setup MongoDB to local system, create database with name 'employee' in MongoDB
- If not able to connect with the above MONGO_URL, try with MONGO_URL=mongodb://localhost:27017
- If MongoDB is already setup, create employee database, provide appropriate database URL in 'MONGO_URL' field in '.env' file, respective database username and password can be provided in 'user' and 'pass' fields present in 'initDB.js' file in application
- Open fresh new terminal in VS Code, navigate inside project directory and run command 'node server.js'
- Have Thunder Client (API client extension in VS Code, easy to navigate and test API's within single platform) installed within VS Code, can use any API client software

- As soon as the server starts and database connection is established, one can,
- Send requests as,
- GET /employees: Retrieves a list of all employees
- POST /employees: Creates a new employee
- GET /employees/:id: Retrieves a specific employee by their id
- PATCH /employees/:id: Updates a specific employee by their id
- DELETE /employees/:id: Deletes a specific employee by their id
- GET /employees/:id/tasks: Retrieves all tasks associated with a specific employee by their id
- POST /employees/:id/tasks: Creates a new task for a specific employee by their id

JSON objects for testing

For Employee Model -

```
{  
  "ID": ,  
  "name": "",  
  "email": "",  
  "phone": "",  
  "hiredate": "",  
  "position": ""  
}
```

For Task Model -

```
{  
  "ID": ,  
  "title": "",  
  "description": "",  
  "duedate": "",  
  "employeeId":  
}
```